

Juha Lappi: Jakta- users' guide **version Sep, 2002**
Recent changes at the end,

Jakta

Jakta is a general statistical program, which contains e.g. linear, nonlinear and non parametric regression, figure plotting etc. A forest management schedule simulator is included now as an extension of its transformation routines. Jakta is operated using command lines, but it contains many tools which make this kind of operation mode more efficient, e.g. commands can be included from files so that a part of the commands lines is reinterpreted, commands can be generated using loop constructs etc. These properties are called here as command generation programming.

If a line ends with '>' then the next line is a continuation line.

Variables

Variable names start with letter A-Z, a-z, or \$ (temporary variables).
 Characters \$.#@ are allowed in variable names.

Special variables

```
n           =   number of the observation
reject      =   variable used to reject observations
               observations with negative value are rejected for
               the current operation
$           =   variable used in write to indicate screen or '*' -format
printlevel  =variable telling how much to print to the screen
               0- nothing 100=all (default)
outlevel=    variable telling how much to write into the default outputfile
               default=0, give the name for file with: open outfile=
outfile     = unit for outputfile, can be used also in write
regf        = variable used to store the value of the regression function
resid       = variable used to store the residual
trace       = variable telling how much of command input is printed
               = 0 ! the lines read from include files are not printed
               = 1 ! when lines from included files are printed, but in ;do loops only for first round
               = 2 ! do loops are printed for each round,
               = 3 ! (or just trace) print also lines bypassed with goto or when including files using starting
address     = -1 ! the default value takes control
keeperiod   !number of subperiods kept in the simulated decision tree
```

Using variables

Using variable lists in commands (later called `v_list`):

```
read x1,y,h,t
read h,x1,-x20,vara,-varx
```

Symbolic names for variable lists:

```
list a=a1,-a10
list c=c1,c2
list d=d1,d2
list h=c,d    ! thus h is equivalent to list c1,c2,d1,d2
```

Giving numeric values for variables:

```
cons var1,-var6=1,2,sin(3)+sqrt(5),4,5,6
```

The one-line `cons` defines ordinary transformation line which is immediately executed.

```

cons
if(c1.eq.2)then
var1=exp(var2*cos(a))
else
var1=1/c+d
end if
/
cons/rep ! recompute cons transformations

```

Printing current values of variables:

```

print v_list
print 'text1',v_list1,'text2',v_list2 !prints text plus values
print 'text' ! prints only text
print/all !prints all named variables
print/name v_list ! prints both names and values of the variables
print/index v_list ! prints also variable indeces (needed for debugging)

```

Printing works also for tables.

A variable can be give also character value which are utilized in formats and file names:

```

text form=('kukuu',f5.1)

```

Table

```

table d(6)=3,2,3*sin(h+4),4,5,6 !defines one dimensional table
table coef(2,4)=1,2,3,4,11,12,13,14 !defines two dimensional table
out=d(2+i*j)+coef(1,k+4) !using tables in transformations
print coef !printing table

```

One dimensional tables are interpreted in matrix calculations as column vectors. In two dimensional tables first index is row number and second index column number.

Continuation lines for table can (and must for very large tables) be given without the continuation line character (>)

Tables can be also defined using any named transformation set (see transformations). E.g.

```

trans tabde
out=i+0.1*j+ran(seed)
/
table a(2,2)=tabde(i,j,out) ! use transformation set tabdes

```

arguments are: the index variables (row, column), and the last argument is the variable in transformations telling the value of the table element.

Variable names can also refer to parameter sets and later also to figure elements (see transformations).

Command generation programming

```

;addr: !address in command file , address can contain only alphanumeric characters
;su('arg1','arg2','arg3') !subroutine type address with associated arguments
incl
Options of incl, only first letter in option is significant (except DELETEME). Several options can be in the same
command.
incl file !Commands are read from file from start to end or ';return'
incl file/adr !Commands are read from file starting from address 'adr'
incl file/su(ar1,ar2,ar3) !call subroutine with arguments
incl/close

```

Current file is closed and then a new file is included. This way included files can be chained (without /c the control would return to the next line in the current file after the new include file is completely treated, which would make no harm if this is last line)

```
incl/DELETEME
```

The current include file is deleted and then the new file is included. The whole option name 'DELETEME' is necessary (to avoid accidental deletions).

```
incl/wait file
```

If the include file is not ready it is waited until it is. If an interface program is using all the time file 'j.inc' to transmit commands to Jakta then this can be done so that the interface is writing always to the end of file j.inc the line:

```
incl/w/DELETEME j.inc
incl/optional jlpinit.inc
```

If the file is available it is included. This will be used by Jakta to obtain possible initialization commands

```
;return !The included file is closed and control returns to upper level
assi 'region'=Savo !Later 'region' is replaced in commands with text Savo
ask 'region'=what is the region for the problem !Asking symbol from terminal
;goto label !goto address ;label:
;if(status.le.2.or.var1.gt.1);goto ad1 !the command after ;if() is ;-command
;if(problem.eq.1)regr y,x1,1 !but it can be also command of Jakta
```

***Note that goto, if and return (without ';') are available in transformations. In transformations the addresses are given without initial ';'.

If variable nper has value 4, then the following command line is changed:

In this line "nper" and "2*nper+1" are replaced ->

In this line 4 and 9 are replaced !the command goes into Jakta in this form

```
;do i=1,nper
volume"i"=volume"i-1"+growth"i"
;enddo
```

```
pause ! pause
```

```
test ! check only command generation. Also ""-sections are interpreted. This way one can test if all include files are available.
```

```
test/no ! End of testing mode.
```

TRANSFORMATIONS

Transformations follow Fortran style. Addresses end with ':' and cannot start with ';'.

Flow of control

```
goto adr1 !Note this is different from ;goto)
```

```
call adr1 !when encountering return control returns
```

```
return !return after the previous call , note this is different from ;return
```

```
exit ! stop executing transformations
```

Example:

```
if(v1.gt.1)goto ad1
call ad2 ! There is no argument passing system
ad1:write($,$,v1)
exit
ad2:write
return
```

Functions:

int, abs, sqrt, log, log10, exp, sin, sind, cos, cosd, tan, tand, asin, asind, acos, acosd, atan, atand, sinh, cosh, tanh, max, min, mod

*Functions sind, -atand have arguments in degrees, sin, -atan have arguments in radians.

*Functions max and min can have any number of arguments.

x1=swap(x2) !change values of the variables

blk(x2,r1,r2,r3,r4) ! if x2<r1 then blk=0, if x2>=r1 & x2<r2 then blk=1, if x2>=r4 then blk=4

linear(x2,c1,c2,c3,c4,y1,y2,y3) !piecewise linear interpolating function through points (ci,yi) evaluated at x2

dot(c1,c2,...,cn,x1,x2,...,xn) ! inner product c1*x1+c2*x2+...cn*xn

sum(x1,x2,...,xn) ! sum x1+x2+...+xn

* Functions linear, dot and sum can nicely utilize lists.

ran(seed) ! random uniform number

rann(seed)! random normal with mean 0 and variance 1

*Seed in ran and rann determines the exact sequence. It must be a variable for which the initial positive value is set with cons command. The value of seed variable is changed in the subroutines into zero. A new sequence can be initialized by giving a positive value to the seed variable again. The random number generators are based on ran1 and gasdec routines of Numerical Recipes book.

densn(mean,sd,x) ! the value of the density function at x for a normal distribution for given mean and sd

desln(mean,sd,low,x) ! the density of lognormal distribution having lower limit 'low'

distn(mean,sd,x) ! the value of cumulative distribution at x, i.e., the probability that the variable $\leq x$.

distln(mean,sd,low,x) !cumulative distribution for log.normal variate with lower limit 'low'

truncn(mean,sd,x0) ! mean of normal distribution truncated at x0

truncn(mean,sd,x0,sd2) ! with additional parameter sd2, the function returns also the sd of the truncated distribution

trunln(mean,sd,low,x0) ! the mean of truncated log-normal distribution, the original lower limit is 'low'

trunln(mean,sd,low,x0,sd2) ! the sd of the truncated distribution returned in sd2

get(\$,n+lag,x1) ! get the value of x1 in observation n+lag (n is the current observation). The first argument tells the data-set., \$ means the current data set. Currently first argument is ignored as data-set object is not yet supported. If the second argument is not legal observation number, function returns 1.7e37 (=missing value). The third argument must be variable stored in data matrix (use make if necessary), if it is not the function returns 1.71e37.

put(\$,n+lag,value,var) ! put the value 'value' in variable 'var' in observation n+lag (n is the current observation). The first argument tells the data-set., \$ means the current data set. Currently first argument is ignored as data-set object is not yet supported. If the second argument is not legal observation number, function puts 1.7e37 (=missing value). The fourth argument must be variable stored in data matrix (use make if necessary), if it is not the function does nothing, but returns value -1 (successful return 0)

lastin(class) ! returns the number of last observation having the same value of variable class as the current observation. Variable class must be stored in data matrix, if it is not function return value -1.

Class mean of variable x for classification variable c can be computed into separate class mean variables as follows (later special routines will be provided, as well as explicit loops):

```
trans
if(n.eq.1)old=-1
if(c.ne.old)then
sum=0
ncl=lastin(c)-n+1
```

```

i=0
loop:sum=sum+get($,n+i,x)
i=i+1
if(i.lt.ncl)goto loop
xm=sum/ncl
old=c
end if
/

```

Arithmetic operations:

The precedence order of arithmetic functions is: ***, **, *, /, + and - (** is integer power).

Matrix calculations

Matrix calculations can utilize vectors and matrices defined with table, or matrix obtained from data with command 'matrix'.

Matrix functions:

```

t(a) !transpose of a
inv(a) !inverse of square matrix a
row(a,b,c) !matrix obtained joining columns of a,b,c, or row vector obtained from scalars a,b,c
col(a,b,c) !matrix obtained joining rows of a,b,c, or column vector obtained from scalars a,b,c
diag(a,b,c) !diagonal matrix with scalars a,b, c on the diagonal
diag(a) !diagonal matrix with row or column vector a on the diagonal
eigen(c,lamda) !computes eigen vectors and eigenvalues of the real symmetric matrix c, the output variable will get the eigenvectors, and the eigenvalues are stored in lamda. c can be also the covariance matrix obtained from corr-command. Only the values on and below the diagonal of matrix c are used.

```

* multiplication

```

a*b multiplies a by b, where a can be scalar or vector or matrix, and b can be scalar, or vector or matrix
a+b, a can be scalar or vector or matrix, and b can be scalar, or vector or matrix
a-b a can be scalar or vector or matrix, and b can be scalar, or vector or matrix
-a negative of matrix a

```

If matrix is multiplied by a scalar then each element is multiplied by the scalar. If a scalar is added to a matrix then it is added to each element.

If addition or multiplication is not defined, then output variable is not changed.

NOTE

If a is a matrix then substitution

```
b=a
```

does not work yet. This can be circumvented using e.g.

```
b=row(a) or b=col(a) or b=1.*a or b=0+a
```

A random vector can be generated e.g. as follows:

```

cons seed=17
trans rangen
eps=rann(seed)
/
table e(6)=rangen(i,eps)

```

New values are obtained by giving table command again.

Data matrix

command

```
matrix m=varlist
```

generates (number of observations) X (size of varlist) matrix m which using observations data. E.g. computation of regression equation can be done either using

```
regr y,x1,x2,1
```

```
or
matrix m=x1,x2,1
matrix yv=y
cons coef=inv(t(m)*m)*t(m)*yv
```

Derivatives:

```
dx,dz,dy=der//x,z,y
f=sin(x)*exp(c*y)/(z*d+y)
```

Splines:

Splines can be generated with group command, or stem curve splines directly in transformations. If group has options /spl/out=Var then the value of the regression spline can be computed with transformation (see group for more details):

```
pred=splin(Var,x1)
```

A stem curve spline is generated with command

```
calle=spl(dvars,hvars) ! calle is name (any) for the generated spline,dvars
is variable list for diameters (in centimeters), hvars is variable list for
heights in increasing order (in centimeters). If the measurement height has
smaller value than first height (e.g. negative value), it indicates that
there are no more measurements for the current tree. This is usefull when
there are different number of measurements for different trees, then one can
use the same lists as arguments, and the dimensions of lists are defined
using the largest number of arguments.
```

The generated spline can then be utilized:

```
d130=spld(calle,130) ! dbh according to the spline
h=splh(calle,d) ! height for diameter d
v=splv(calle,h1,h2) ! volume between heights h1 and h2
```

Less oscillating taut splines are generated with tautspl function

```
it=tautspl(gamma,xval,yval) ! argumen order may change later
```

where xval is list containing x-values and yval contains yvalues, and gamma is the flexibility parameter. gamma=0, ordinary cubic spline, with values between 0 and 3 additional knot points are addes in order to avoid oscillation, for gamma values between 3 and 6 knot points are added also in intervals with inflection point. If any later x-value is smaller than the first then this and additional points are ignored. Tautspline is used with function

```
y=taut(it,x) ! computes the value
y=taut(it,x,ider) computes the value of the ider'th derivative
```

Tautsplines can be converted into format used in stem curve splines using function 'calles':

```
js2=calles(it)
```

Thereafter spld,splh,and splv can be used. Note that spld is doing the same thing as function taut.

write-function in transformations:

```
write(unit,formatvariable,v_list)
```

where unit is defined with open command, e.g.

```
open unit=filename or
open/replace unit=filename
```

Predefined unit variable \$ implies that writing goes to the screen.

Formatvariable is given character value using text command. Format must obey Fortran standard. Using variable '\$' as the format variable implies writing with Fortran '*' format. Examples:

```
text form=('kukku',f4.1,' kuu2',f9.1)
open ouf=out.txt ! open/replace ouf=out.txt does not generate version
numbers
trans
write(ouf,form,x1,x4)
write($,form,x1,x4)
write(ouf,$,x1,x4)
write($,$,x1,x4)
close ouf ! now you can print or edit file during the Jakta session
```

Using file variable 'outfile' directs writing to the same file where also other commands of Jakta write automatically output if variable outlevel is given nonzero value.

There is special version of write for printing the decision tree in the simulator, see simulator.

*There will be in future read commands available within transformation section, e.g. for reading parameter sets

Making figures within transformations:

In future there will be several ways to make figures within transformations. In order to give an idea what will be available (and to ask good suggestions), there are now three plotting functions.

```
initfig(x,y) ! initialize a figure
draw(x1,x2,x3,y1,y2,y3) ! draw line through any number of points (x1,y1),...
showfig() ! output figure into figure window, and wait for a click of the mouse
```

The third argument in draw can be used for defining an option.

```
draw(x1,x2,x3,y1,y2,y3,-1) ! points are sorted with respect to x before
drawing
```

```
draw(x1,x2,x3,y1,y2,y3,c) ! points are plotted using symbol with ascii code c
#=35, *=42, +=43, .=46, A=65, a=97
```

Special commands in transformation definitions:

```
clear ! clears all transformation definitions
edit ! Only transformations defined after trans command. Writes current transformations into file
'trans.tmp', waits until you have edited the file (e.g. with wordpad, or notepad), then if you type anything jakta
reads the edited transformations.
```

Using transformations in jakta

```
cons ! gives values directly for variables
```

```
cons a1,-a4=sin(1),2,3*4,sqrt(a)+b
cons
a=b+c
...
/
```

```

cons/repeat    ! recomputes cons transformations.
cons/add      ! add transformations to previous cons-transformations

trans/read    ! defines transformations done when reading data in (must be defined before read command)
(transformations)
/
trans        ! defines transformations when using data
(transformations)
/

```

All trans-transformations are done for each observation each time data is went through, i.e. output variables are not added to the data set. This means that is you use a variable as output variable which is already in the data set, the original values remain unchanged.

Variables made by trans-transformations can be added to the data set using make command:

```

trans
x3=x1+x2
/
make x3 !this adds x3 into the data set, but the transformation still remains until you remove it e.g. using clear.

```

Named transformation sets can be generated as follows

```

trans trans1 !generates transformation se with name trans1
x3=x1+a      !ordinary transformations
...
/

```

These transformation are computed once using command

```

calc trans1

```

Named transformations can be used to generate tables (see table) and draw functions (see draw-command).

If transformation definition is closed with '//' instead of '/' then the transformations are computed directly.

If trans command is given with option /add then the following transformations are added to the previous set.

Commands for reading data

The format of input data:

```

form *      ! list-directed, default
form b      ! binary
form (12f2.1,5x,f5.0) !fortran format, note all variables are real
file data.dat !defines the data file for input (not needed for read/terminal)
read        ! read variables from the the data
read x1,x4,y1,-y14 ! reads the variables
read/ter v1,-v4    ! reads variables from the terminal (input stream)
read/init      !empty old data first
Variables whose names start with '$' are not stored in data set.

```

There can be several file - read commands. If read is given with option

```

read/mer v_list
new variables are concatenated to previously variables
.

```

Commands for writing data

```

ofor      ! format when writing data, default *, binary format b (file only)
v_list    ! writes variables into the terminal, at most 100 obs written
write/file=data.out v_list ! writes variables into file

```

```
write/form=b/file=out.dat v_list ! format can be given also in the same line
write/file=data.out/close v_list ! file is closed after writing
```

If the output file exists then Jakta asks if it can be replaced. If write is given with option /replace then it replacement is not asked.

Reorganizing data

Assume that in the original data there are for each tree several observations, each observation containing some tree variables, plus variables dm and hm for describing the measured diameter + the measurement height. We may want to reorganize the data so that the new observation will be tree, and different diameters and measurement heights will become separate variables (this form is assumed e.g. in the stem curve modules). This can be done using join command. One needs first specify variable lists to inherit measurements. Assume that there at most 25 measurements per tree, and variable tree is index for trees (any variable having different value for different trees can be index variable)

```
list h=h1,-h25
list d=d1,-d25
join/expand=hm,dm/into=h,d/key=tree
```

If there less measurements for one tree than variables in the into-lists then value -9 is given for the extra variables. If there are not enough variables in the into-lists, an error occurs, but the data structure remains unchanged.

Controlling the output

All commands will print results to the screen or outputfile, or both. How much is printed and where is controlled with variables 'prinlevel' and 'outlevel'.

```
0 = print nothing
1 = print only one line indicating if the commands was successfully done
2 = print standard output
100 = print everything (also some test result useful in debugging)
1000=higher level debugging
```

Outfile can be given name using 'open outfile='. If name is not given, the output goes to file fort.51. If file with this name already exists the old is automatically deleted (by the system) .

Note: ;trace controls how much of the input is printed

Plotting figures

In the same figure one can collect several elements. This means that the current figure needs to be explicitly cleared with clear-command.

```
Symbol / means that a line is drawn to connect points.
clear ! clear the current figure
x var1 ! var1 is the x-axes
x var=min,max ! defines also the range
x var=min,max=dx ! dx defines the difference between ticks.
plot v1,v2,v3=*#o ! plots variables using symbols *#o
plot v1,-v3 ! plot using previously used characters (default point)
plot v1,v2=*#=0,100 ! give the also range for y-axis
plot/noshow ... ! augment plot but do not show it
show ! shows the current figure
show/mat !write the figure into text file fort.74 in format which can be
```

copied into Mathematica so that a Mathematica module can plot the figure (ask J. Lappi for the module)

Drawing functions

Define x-axes with range. If new variable name is wanted, create variable with cons command (e.g.: cons xax=0)

```
x xax=0,10=1
trans
f=sin(xax)
/
draw f
```

Command:
draw regf
will draw the regression function.

draw/cons f ! use cons-transformations to compute f (cons -transformations must be defined in whole transformation paragraph, not in one-line cons-command)

draw/trans=tf f ! use named transformation set 'tf'

*See also section 'Making figures within transformations'.

grouping data

group command can be used to group data, and produce both figures of group means and tables, and to compute regression splines (and store these so that they can be used in transformations). Command x is used to define the variable which determines the grouping. If dx is not given then the range is divided into 10 groups.

```
group v_list ! print a table of class means for each variable
group/min=5 ! combines groups so that each group has at least 5 obs.
```

with other options, group operates on single variable

```
group(/options) var
  /fig ! add group means, standard deviations, standard errors of means into figure
  /fig/b ! do not draw standard errors
  /fig/noshow ! do not show the figure (just augment it)
/spl computes a regression spline, suboptions
  /reg=fvar ! store the regression function into variable fvar
  /reg=fvar,resvar ! store in addition the residual into resvar
  /reg=fvar,resvar,res2 ! in addition store the squared residual into res2
  /out=soutv ! store the spline parameters into variable soutv the spline can then be used in transformations as
  splin(soutv,x) where x is the argument variable
  /constr=(constraints) where constraints define values or constraints of the regression function or its first or
  second derivative for some arguments, e.g.
  /constr=f'0=0,f'300=0,f''300=0 ! f(0)=0, f'(300)=0, f''(300)=0
  /constr=f0-f200>0,f'0>0 ! f(0)-f(200)>0 f'(0)>0
  /knots=0,30,75,150,225,260,300 !define knot points, first and last must be compatible for a
  reasonable range, default is that the range of x-axes is divided into 4 interval, i.e. the number of knots is 5.
  /knotticks=1 !draw ticks to the top of figure to indicate the positions of ticks, the value is how many
  percent is the length of the tick from the y-axis.
  /order=2 !parabolic spline
  /order=3 ! cubic spline (default)
  /nodraw do not draw splie function
```

Note: /spl option by default generates plot of the regression function, /fig option only tells if error bars are plotted

Statistical commands

```

stat v_list ! computes means, minimums, maximums, and sd's.
options of stat
/mean means are stored in variables with names as mean%x1
/sd sds are stored in variables with names as sd%x1
/var vars are stored in variables with names as var%x1
/min mins are stored in variables with names as min%x1
/max maxs are stored in variables with names as max%x1
corr v_list ! computes correlation coefficients
corr/out=obj v_list !stores the correlation matrix under name obj
corr obj print the correlation matrix object (generated by corr/out or read
from file using unsave). Variable names are not printed because the
corealtion matrix object does not yet store variable names or variable
indices (variable indices may be associated with different variable names
when unsaving objects).
regr v_list !computes ession equation when the first variable is regressed on the others
regr v_list=v_list2 ! stores the regression coefficients into variables of v_list2
regr y,x1,x2,l ! variable l is constant with value one, the coefficient is thus the intercept
regr/corr !correlations of parameter estimates are printed
regr y,x1,x2,l=a,b,c ! stores parameters into variables a,b,c
regr y,x,l=a,b,rmse ! if there are more parameter variables rmse is stored into the last

```

Regression automatically creates variables regf and resid for the regression function and residual

Nonlinear regression

First define the nonlinear regression function and the derivatives of the parameters:

```

trans
da,db,dc=der//a,b,c
f=a*x**b+c ! note b is nonlinear parameter , other parameters are linear
/

```

Give initial values directly:

```
cons a,b,c=2,1,0
```

Or you can use linear regression to get initial values for linear parameters:

```
cons b=1
```

```
regr y,x,l=a,c !stores parameters into a and c
```

Thereafter (or before) define parameters to be estimated, and the derivatives

```
para a,b,c=da,db,dc
```

Do estimation:

```
gest y,f
```

If convergence is not achieved, give gest again. The default number of iterations is 6, you can change this

```
gest y,f=10 ! do iteration 10 times
```

```
gest/corr y,f ! print correlations and standard errors of parameter estimates
```

Mixed models

Variance components of a group of variables can be computed with command varc

```

varc/class=var1 var_list
where var1 is grouping variable. Varc can have following additional options:
/sdm compute also the standard error of the simple arithmetic mean
/mat write into file fort.74 data for making a error bar type figure for the arithmetic mean
/varb for each variable store between variance into variable varb%variablename
/varw store within variance into variable varw%...
/mean store simple arithmetic mean into mena%...
/cov compute also covariance components
/out=covb,cov1,-cov7 !store between covariance matrix and within matrices ,
the number of within matrices indicates that the argument variables are grouped into that many groups

```

Initialization

Init ! clears everything, and provides a new fresh start (may not work always !!).

Stem form modeling

A multivariate nonparametric stem form modeling, can be done using standard Jakta commands. There are some special command for using the devlopde model. The method si described in more detail in manuscript Lappi: a multivariate nonparametric stem curve prediction system'. The model development has the following stages:
NOT UP TO DATE

Generate stem curve data in polar coordinates, e.g. as follows if the initial data are relative height diameters

```

**relative height diameter variables:
list diam=d1,d2p,d5,d7p,d10,d15,d20,d30,d40,d50,d60,d70,d80,d90
**corresponding percentages
table pros(14)=1,2.5,5,7.5,10,15,20,30,40,50,60,70,80,90
**angles for polar coordinates
table ang(12)=0.25,0.7,1.5,3,5,8,14,21,31,41,56,72
**compute sins of the angles to generate table
trans sindef
if(i.le.12)then
sinout=sind(ang(i))
else
sinout=1.
end if
/
table sins(13)=sindef(i,sinout)
**relative heights
list ht=h1,-h14
trans/read
**initial data had height in meters, in splines they are suem in centimeters
hcm=100*h
**gnerate relative heights
;do j=1,14
h"j"=pros("j")*h
;enddo
** define spline
calle=spl(diam,0.4,ht,hcm)
**compute diameters at specified angles
;do j=1,12
da"j"=spla(calle,ang("j"))
;enddo
**height is the last 'diameter', height is in meters
dal3=h
/

```

Thereafter Jakta is used to compute the regression splines when each dimension is regressed in turn on each other dimension, and the relative squared residual (variance function) is described with regression splines with respect to the original independent dimension:

```

**define first reasonable upper limits for dimensions.

table maxi(13)=70,60,55,50,50,45,40,35,30,25,18,10,30
;stem:
**do not print the group-tables
cons printlevel=0
** loop for independent variables
;do i=1,13
clear
print 'alku,i ',i
x da"i"=0,maxi(i)=0.1*maxi(i)
*main loop makes mean splines and variance splines
;do j=1,13
**construct spline
;if(j.ne.i)group/min=5/fig/noshow/spl/reg=f"j",res"j",>
ress"j"/out=s"i"#"j"/knotticks=1>
/constr=f0=0,f' 'maxi(i)=0 da"j"
;end do
show
**regression of variabel on itself is not done, but the
** corresponding 'spline' variable is made so variables lists can easily used
cons s"i"#"i"=0
** first save opens the parameter file
;if(i.eq.1)save/outfile=stem2.sav/replace s"i"#"1",-s"i"#"13
;if(i.gt.1)save s"i"#"1",-s"i"#"13
clear
;do j=1,13
;if(j.eq.i);goto ohi
trans
clear
**relative sqaured residual, 10000 is just to get nicer numbers
relress=10000*ress"j"/(f"j"*f"j")
/
**secvond degree splies for variances
group/min=6/fig/b/noshow/spl/order=2/reg=fv/out=ss"i"#"j">
/knotticks=1>
/constr=f'0.12*maxi(i)=0,f'0-f'0.12*maxi(i)>0,f0-f0.35*maxi(i)>0,>
f'0.35*maxi(i)<0,fmaxi(i)>1,f'0.9*maxi(i)<0,>
f'maxi(i)=0,f'0.9*maxi(i)-f'maxi(i)<0>
/knots=0,0.12*maxi(i),0.25*maxi(i),0.35*maxi(i),0.6*maxi(i),0.9*maxi(i),maxi(
i) relress
trans
clear
**standardized residuals
sres"i"#"j"=res"j"/(f"j"*0.01*sqrt(fv))
nel"j"=sres"i"#"j"***2
/
make sres"i"#"j",nel"j"
;ohi:
;end do
show
cons sres"i"#"i"=0
cons nel"i"=0
cons printlevel=2
stat sres"i"#"1",-sres"i"#"13,nel1,-nel13
**compute correlation matrix for each independent variable

```

```

MIXED MODEL NOW
corr/out=cor"i" sres"i"#1,-sres"i"#13,nell1,-nell13
cons printlevel=0
cons ss"i"#1=0
save ss"i"#1,-ss"i"#13
** end of independen varaible loop
;end do
** save correaltion matrices
save/close cor1,-cor13
cons printlevel=1
;return

```

To use the estimated model in a seaprate run, it needs first to be loaded:

```

unsave/infile=stem2.sav s1#1,-s1#13
unsave ss1#1,-ss1#13
;do i=2,13
unsave s"i"#1,-s"i"#13
unsave ss"i"#1,-ss"i"#13
;end do
unsave/close cor1,-cor13

```

```

**the angles table needs to be definde
table ang(12)=0.25,0.7,1.5,3,5,8,14,21,31,41,56,72

```

The model is defined with stem-command:

```
stem/angles=ang/mean=s/variance=ss/correlation=cor/out=pine
```

It si assumed that the angle indices of mean and variance splines and correlation matrices are as above. '/out=' gives the name for the model. There can be several coexistent models.

The defined model can be used in transformations using two functions

```

** compute model for the tree,
**pine= name of the model
** then comes diameters and then heights, first diameter is used as the independent variable
** here it is assumed that dbh and height are measured, heights need to be in centimeters
iout=jstem(pine,dbh,0,130,height)

```

TREE

**thereafter one can generate stem curve spline using jspl-function

```
js=jspl(tree,0)
```

or taut spline using

```
js=jspl2(tree,0,gamma)
```

where gamma get flexibility parameter, gamma=0, ordinary cubic spline, with values between 0 and 3 additional knot points are addes in order to avoi oscillation, values between 3 and 6 knot points are added also in intervals with inflection point. Taut-splines ar used with taut-function

The argument will have several interpretations. Currently

0= ordianry stem curve spline

10= the curve on would predict using the independent dimension alone

ip=jpolar(tree,da1,da2, ,da13) !after calling jstem, the diameters at polar angles + height will be stored in the argument variables. The output-variable will get nonzero value if the number of arguments is not number of knot angles +1. Currently jspla does not work properly after jstem and jspl because the solution is already in knot angles and the rounding errors will often force the solution at predetermined angle to be at wrong side of the knot angle. If there are angles+1 more arguments, then the standard errors are stored in these variables

JSCOVW

SIMULATOR

Jakta contains also a simulator, which can produce data for JLP. As there will be no build-in growth or treatment models, it might be better to say that there is a simulator language included and not a simulator. I call it anyhow a simulator. The simulations can be done both at stand and tree level. It is also possible to combine both stand and tree level simulations in the same run. E.g., it is possible that tree level simulations are done only for first periods and later periods are simulated using stand level variables, or one can simulate protection areas at stand level and timber production areas at tree level, or after regeneration simulation can be at stand level up to certain age. 'Stands' can actually be fields or lakes etc.

Stand data are read using file, form, and read commands of jakta. Transformations can be done with trans or trans/read.

Simulator section is started with jakta-command jlpsim.

If there are tree data available then there needs to be commands:

```
treev variable_list
treed tree_data_file
```

If treed is given with option /own then the reading is done with owsrea-subroutine controlled by the user.

The default file type for tree data is sequential ascii file, and the default format for reading tree variables is Fortran *-format. If these are not ok, then the format must be given with command treef-command.

```
treef b ! binary data
treef (2f4.0,5x,5f7.1) !fortran format
treef own ! file opening with special owope-subroutine.
```

It is currently assumed that for each stand there is one record where there is first number of separate trees, and then for each tree there are the values for tree variables. If each tree is associated with a frequency telling how many similar trees there are, then this frequency variable is just among tree variables.

If not all simulated variables are stored use keep or drop:

```
keep var_list ! what simulated variables are stored
drop var_list ! what simulated variables are dropped
```

The simulator generates files for JLP if there is command

```
out name
```

The files generated are name.inc, name.xdb, name.cda. The data can be loaded into JLP using command incl name.inc. File name.xdb contains binary xdat data, and name.cda contains cdat in ascii format. If the output files already exist, then Jakta will ask if they can be replaced. If the out command is given in form:

```
out/replace then files are replaced without asking
```

The simulator is then defined in section between commands

```
simu i=1,nper
...simulator
endsim
```

where 'i' is freely chosen index variable for periods, nper a numeric constant or a variable or transformation telling the number of periods. This loop over periods works basically as ;do-loop in j-language. Within the loop there are treatment sections telling when a treatment is applied and what then happens to stand and tree variables. Treatment sections start with

```
name_of_treatment::if(condition_when_treatment is applied)
```

Section ends when the next section starts or at 'endsim'. Named constants are created for each treatment, so that these variables can be utilized in simulation transformations, or later in optimization. It is recommended and it may become necessary that the first treatment is given without any condition, and usually it is no let grow option) e.g.:

```
grow :
```

The condition for treatment is any logical statement using stand variables and the current period (the index variable). The condition can use also random numbers.

Stand development is described using ordinary transformations. Index variable and double quotes sections are used to define different variables for different periods and the dependency of next period on the previous ones.

In simulations the schedule generation tree is traversed in depth first order. If there are three treatments, grow, thin, and clearcut, treatments conditions are tested and possible treatments are applied in order (if initial part is empty it means that the same treatments are applied as in the previous schedule)

```
periods
1      2      3      4
grow   grow   grow   grow
                        thin
                        clear
                        grow
                        thin
                        clear
                        grow
                        thin
                        clear
                        grow
                        thin
                        clear
                        grow
etc.    thin   grow   grow
```

All variables keep their values until explicitly changed. This property and the simulation order can be utilized by using variables calculated for the previous treatment. Common transformations done initially for each stand can be defined in grow section.

nextperiod : defining a treatment sequence in one section

A traditional way to define a treatment program is so that one defines the whole sequence of treatments at once, e.g. thinning at age 50 and 80 and clearcut at age 100. It is possible but not very easy and natural to define such sequences using the treewise generation (one should define indicator variables telling if we are in the intended sequence).

This kind of sequence can be defined using command nextperiod. A simulator with nextperiod is, for instance

```
simu i=1,nper
```

```
prog1::if(age"i".ge.50.and.age"i".lt.60)
call thin1 "i"
call growt"i" !age will be 60
nextperiod
call growth"i+1" !age will be 70
nextperiod
call growth"i+2" !age 80
nextperiod
call thin"i+2"
call growth"i+3" !age 90
nextperiod
call growth"i+4" !age 100
call clearcut"i+4"
prog2:: .....
```

During simulation, the nextperiod command increases the period number, updates also the index variable 'i', that is it generates a node into the simulation tree. The implementation of 'nextperiod' automatically takes care that the

node has no sisters. But the nextperiod does not increase the period index ('i' above) during the simulator definition. That's why after nextperiod the proper variables are accessed by using "i+1!", "i+2" etc. Note that the prog1 sequence does not define necessarily treatments to the end of planning period, i.e. after the clearcut the continuation is determined by other treatments. The other treatment could take over already earlier, if after having thinnings at ages 50 and 80, the clearcut is left to a different clearcut treatment with its own entry condition.

Store/load

It is possible to store the values any stand or tree variables for later use using command

oldval=store(varlist) !where oldval will be the name for the set of the stored values

These values will be return to variables using command

load(oldval)

If the old values are needed when testing treatments, then the treatment header line must be given in form

thin::load(oldval)::if(.....

Later one can load several sets with the same load-command but now only one set. Note that as you can really use only stand variables in testing treatments it may not be reasonable to store tree variables in the same set which is needed for treatment testing as the load command would cause copying of tree variables even if they are not needed if the treatment test is not passed.

There is no predetermined assumptions what is the exact timing of treatments within periods. The simulator programmer must take care of this. If treatments are in the middle of periods, then the simulation order can be utilized by using the logic:

```
grow : :
(grow stand to the middle of period, use separate middle of period variables)
(grow the rest of period)
thin : : if ( )
(make thinning based on the middle-of_period_variables)
(grow the rest of the period)
```

Cutting off branches

There are two way how the standard simulated decision tree can be modified. First, if `exit` command is executed at any node then the subsequent nodes are not generated. All variables associated with later periods have values they have obtained earlier. Note that in ordinary transformations `exit` command just will end transformation generation.

Second, if variable `keeperperiod` has been given value with `cons` command which is smaller than number of periods in `simu` command then the simulator travers the decision tree as usually but a schedule is generated and output into JLP file as all nodes below `keeperperiod` level have been visited. Using this property the simulator can do directly e.g. stand level optimization after `keeperperiod` level. E.g. the net present value (2 and 3 % interest) of standing inventory can be computed as follows. Assume that the main planning horizon is 5 10-year periods and the maximum rotation is 15 periods and assume that there are no local optima for rotation and rotation for 2% interest is longer.

```
simu i=1,15
grow::
if(i.eq.5)then
fnpv2=-1000
fnpv3=-1000
end if
...
clearcut::if(...)
....
$npv=income/1.03***((10*(i-5)) ! faster: "1.03***((10*(i-5)))"
```

```

if($npv.gt.fnpv3)fnpv3=$npv
$npv=income/1.02**(10*(i-5))
if($npv.gt.fnpv2)fnpv2=$npv
if($npv.lt.fnpv2)exit ! optimum has been passed, no need to generate simulate further

```

Tree loops

If tree variables and tree data file are given in 'treev' and 'treed' commands, then in the simulator there can be any number of loops over trees in format:

```

for j=1,ntrees
(transformations using tree and stand variables)
end for

```

NOTE ! Temporary variables (name starting with \$) cannot be tree variables. They can be used within tree loops as output variables but their values will not be retrieved if they are used in later tree loops.

Also any tree variables keep their values until explicitly changed. Thus one can apply the same logic as in simulation with stand variables only, i.e. one can simulate in grow-part up to the mid-period, and utilize these variables in other treatments.

The simulator does not assume any fixed tree variables. Usually there is a variable telling how many similar trees exist in the stand. Cutting of trees can then be done by decreasing this variable.

Tree loops can be done in ascending or descending order with respect to any tree variable by computing first ordering indexes with sort function:

```

orderd=sort(d)

```

This will actually generate a tree variable where the first value tells what is the number of smallest tree with respect to variable d, etc.

The sorting can be done in descending order using

```

orderd=sort(-d).

```

Later sorting can be defined in terms of function of tree variables (e.g. `sort(d**2*h)`), but now the argument must be a tree variable.

Tree loops are done in the order defined by the sorting if the loop is defined as:

```

for i=1,ntrees/dorder

```

The order is reversed with

```

for i=1,ntrees/-dorder

```

Thus `order=sort(-d)`, for `i=1,ntrees/-order` is equivalent to `order=sort(d)`, for `i=1,ntrees/order`

New trees can be generated by giving or computing as a function of stand variables a value for a variable 'newtrees'. Thereafter there can be loops over new trees in format

```

for j=1,newtrees
....
end for

```

New trees are automatically merged with old trees at the end of treatment section. So there can be several loops over new trees and old trees in any order. If one wants to merge new trees with old trees so that ordinary tree loop would include also new trees this can be done using command:

```

add newtrees

```

Sorting can not yet be done for newtrees before they are added to the tree set.

Note that as the transformation include random number generation, it is possible to make stochastic simulations. The main problem with stochastic tree level simulations is that then one can not properly use trees presenting different frequencies of trees in the stand.

Tree functions

The following functions compute sums of variables over all trees. Note that it is currently not possible to make tree loops within tree loops.

```
output=tsum(treevariable) ! sum of variable over trees
output=wtsum(weight,treevariable) ! weighted sum of treevariable.
output=tsumgt(treevar1,treevar2,var3) ! computes the sum of treevar1 using trees for which
treevar2 is greater than var3. If this is used outside tree loop, var3 can variable having constant value or e.g. mean
diameter. If tsumgt is used within tree loop then var3 must also be tree variable, and usually it is the same variable
as treevar2, but var3 is the value for the current tree within the loop. Often all three variables are the same, but it
can be that treevar1 is e.g. basal area and treevar2 and var3 is height.
Thus output=tsumgt(ba,height,height) computes the sum of basal areas of all taller trees than the
current tree.
output=tsumge(treevar1,treevar2,var3) ! as tsumgt but condition is ≥.
```

```
output=wtsumgt(weight,treevar1,treevar2,var3) ! as tsumgt but computes weighted sum
output=wtsumge(weight,treevar1,treevar2,var3) !condition is ≥
```

Weighted mean can thus be computed as follows:

```
ws=tsum(freq)
vs=wtsum(freq,var)
wm=vs/ws
```

Note that currently in these tree sum functions the arguments must be variables (define constants for var3 with 'cons'), and the functions cannot appear within other transformations, i.e., the format must be:
output=func(argumentvariables).

Note. After the sorting of trees is available, tree functions used within tree loops are not really necessary, and they are less efficient and less general than using sorting and explicit cumulative sums within tree loops done in the sorted order, and thus they perhaps will be dropped from Jakta.

Subroutines

Shared command sections (trivial subroutines) can be used in the simulator as follows:

Period specific subroutines used for different treatments must be defined just before endsim after line 'subroutines'

```
e.g.
simu i=1,4
grow::
..
call sub"i"
...
thin::if(age"i".gt.40)
call sub2"i"

subroutines
sub"i":v"i"=5+v"i-1"
...
return
sub2"i":v"i"=5.5+v"i-1"
```

```

...
return
endsim

```

Common transformations which are done after a complete schedule is generated can be define within 'epilog' paragraph which must come immediately after 'endsim'

```

...
endsim
epilog
(transformations)
/

```

In epilog there can be subroutines as in ordinary transformations.

Simulations are done if first command after endsim or epilog section is 'go'.

If this is not stored in the include files you can give this from terminal after checking that everything is ok.

special write-function in the simulator

If the format used in a write function starts with 's' or 's0' (and is followed by a legal Fortran format) then the writing goes to a special character buffer whose length is determined by the first argument in the write function (max 256). The buffer is divided into (number of periods)+1 sections. The number of periods is usually the upper limit in the simu loop but if `keepperiod` is used to truncate the tree then number of periods is equal to `keepperiod`. If the format starts wit 's0' then writing goes to first section, and if the format starts just with 's' then writing goes to section (current period)+1. The buffer is printed when write function has only unit number and format arguments (format argument has been assigned text starting with 's', other part of format is ignored). The most natural place for this final printing is in the epilog. For example:

```

text sf=s(f2.0,f5.2)
text sf0=s0(f6.2,f5.1)
simu i=1,5
grow::
...
if(i.eq.1)write(100,sf0,ba#0,age#0) !length of buffer is 100
write(100,sf,grow,ba#"i")

thin::if(...)
...
write(100,sf,thin,ba#"i")
endsim
epilog
write($,sf)
/

```

Summary of the simulator

The simulator has thus the following structure:

```

jlpsim !start simulator
treev ! treevariables
treed ! file for tree data
keep ! varaibles stored
drop !varaibles dropped
out ! output files for JLP
simu i=1,nper

treatment::if(condition)
...
for j=1,ntrees !loop over trees
end for

```

```

newtrees= !generate new trees
for j=1,newtrees !loopover new trees
end for
add newtrees !add new treesto tree loops
subroutines !subroutines
sub1"i":....
return
sub2"i"
endsim
epilog !common final transformations
(transformations)
/
go/nogo ! is simulations are done or not

```

Efficiency considerations

To increase the efficiency of the simulation one should compute as much as possible in the simulator definition phase. Thus if you use constant 3.14/40000 in simulator, give symbolic value for this using cons command, or use in as "3.14/40000" so that the proper value is generated when the command line is read in.

If transformations are needed only for first period use construct:

```

simu i=1,nper
;if(i.gt.1);goto ohi
...
;ohi:

```

This is more efficient than if(i.eq.1)then ... endif structure.

JLP

Jakta contains now a test platform for the new jlp. Initially this is a simplified version of old JLP (for all properties here see Lappi 1992: JLP A linear programming package for management planning), but it is using matrix routines which will be used in next lp routine, which will actually be just a set of functions of the new software. The syntax of the jlp command is:

```
jlp/xvar=i1,d2,d3/cdat=test.cda/cvar=ns/rhs=low/rhs2=up/zcoef=zt/zcoef0
```

The command takes the current data as the x-data of the problem The data are read in as usually (using form, file and read commands). The options have the following meaning:

xvar=varlist

gives the x-variable of each problem row, first variable is the x-variable of the objective row. Thus it is assumed that the problem is given using 'one x formulation'. If there is a linear combination of x variables in the original problem then we must make using transformations a new variable for this combination. E.g. if we have $i2-i1$ in the problem, then we must first make

```

trans
i12=i1-i2
/

```

Then we can use i12 as the only variable in the row. If there is no x -variable in a row, then use 1 as the x-variable of the row.

cdat=filename

gives the file where are c-variables. The file must be text file which is read using *-format.

C-variables are given by

cvar=varlist

In varlist there must be variable 'ns' which tells how many schedules there are in each unit. The sum of 'ns' must agree with the total number of observations in data.

Lower and upper bounds are given by

```
rhs=table1
rhs2=table2
```

where table1 and table 2 are vectors defined earlier. Both rhs- and rhs2- tables must be given. The number of elements in rhs and rhs2 is one less than number of variables in xvar. E.g. for the command line before, these vectors could be defined:

```
table low(2)=0,0
table up(2)=0,0
```

Coefficient matrix for z-variables is given by:

```
zcoef=zmatrix
```

and the coefficients of the z-variables in the objective row is given by

```
zcoef0=objz
```

where objz is a vector having as many elements as there are columns in zmatrix. Currently z-variables have no names.

For testing purposes, jlp command generates file fort.16.

Recent changes

2002

September

:nextperiod : defining a treatment sequence in one section of the simulator

/noshow option in plot

When jlp sim does generate JLP files with 'out' command then new versions of existing files are no more generated. If output files exist, then Jakt asks if they can be replaced. If 'out' is given with option

```
/replace
```

then replcement is not asked.

Similarly in 'write' command new versions are not generated, and asking of replacement can be bypassed using 'replace' option.

function 'calles' which converts taut spline into format used in stem curve splines

new options in draw function in transformations

jlp command is added, this does linear programming, see above.

June 2001

Funtions for normal density, lognormal density, distribution function of normal distribution, and of log-normal distribution. The mean and sd of truncated normal and log-normal distributions. Search: densn

Options /mean /sd /var /min /max added into stat command make new variables where the statistics are stored

```
read/init empty old data
```

```
open/replace nuf=file ! replace old file if it exists
```

March 2001

Function get and lastin are provided for handling e.g. data in classes and lagged variables..

February 2001

Some bugs corrected

eigen function in matrix functions, compute eigenvectors of covariance matrix of given matrix

regr/corr and gest/corr prints correlations of parameter estimates

Possibility to store parameters and rmse in regression

tautspl and taut functions can be used to generate and use taut splines

jspl2 get taut splines in stem form modeling

draw-command: function can be defined in cons-transformations, or in named transformations.

October 2000

join-command: Merging several observation into a new observation so that some variables in the original data are put into lists of variables. E.g. original data consists of trees and for each tree there are several diameter measurements, each measurement forming an observation. With join one can reorganize data so that trees are observations, and different measurements are different variables. search : Reorganizing data

June 2000

All tree variables are now connected to some specific period, and an output variable in tree loops can appear only for one period. Thus it is important to bypass transformations for certain periods using ;if();goto construction and not if()then construction. If an tree variable is output for several periods, an error occurs. For plot variables there is no such strict connection to periods because the number of plots cannot vary and the plot data section remains in order even if one stores later plot variables under the same name (if just the depth first simulation order is taken into account when defining the simulator).

Thus updating the initial information for stand using the growth-section of first period is not logical any more because these outputvariables are associated with first period (if there are no new or deleted trees there may be actually no problems). But one can now do initial transformations within prolog-section, which is before simu t=1,nper section:

```
prolog
(transformations for initial state, can include plot , tree, newtree loops and duplication of trees etc.)
/
```

Output tree variables of prolog are just associated with the initial state and work as tree variables read from data. In principle this prolog section can be used to define an ordinary stand calculation system, but this is not yet tested without simulator section.

It is now possible delete trees from tree list using command:

```
delete/fr
```

Where fr is a tree variable for the current period; trees having fr equal to zero are deleted (usually fr is number of similar trees but it can be also just an indicator for trees to be deleted). If there are new trees generated before delete these are automatically first added to tree list (and thus no newtrees loops are not thereafter possible). It is important to delete trees only after all references to the variables for previous periods are done, because after deleting or adding new trees the links to previous periods do not work (this is not yet checked so there will be no error messages, just absurd results).

It is now possible to duplicate trees into two groups which develop differently (e.g. healthy /sick trees). There are two ways to duplicate trees depending if we want to duplicate trees at the beginning of new period or after letting them grow similarly for some time first. Duplicating trees at the beginning of period can be done using loop:

```
for i=1,duplicates
end for
```

where this loop can be within a plot-loop. Within this loop all input variables get their values from existing trees (variables are for the same or previous period), and outputvariables are stored for new trees. The end result is similar as one would get from newtrees loop where value of newtrees-variable is the same as ntrees-variable. Thus after duplicates-loop there can be further newtrees loops modifying the new trees. There can be further duplicates loops (this is not yet tested).

The other way to make duplicates is command
duplicate

which just copies all tree variables for current period into the newtrees section of data. Thereafter these can be modified in newtrees loops.

Later it will be possible to duplicate specific trees, currently this can be done first duplicating all trees and then deleting those new trees which are not needed.

If there are plots in stands, then variable 'alltrees' tells what is the total number of trees in the stand and variable 'ntrees' tell what is the number of trees for each plot (and can be used only within plot loops).

Early June

Some bugs in command programming corrected

/noshow option for group
pause waits now mouse clicking in the jakta-io window
pause text -prints the text first

Both pause and plotting of a figure into figure window are now waiting for mouse click. There is now a reserved variable 'keystate' which will get value 0 if the response was plain mouse click, value 1 if shift key was pressed simultaneously, and value 2 if Cntrl key was pressed, and value 3 if both shift and Cntrl keys were pressed.

;return can be now given in form
;return/jaktacommmand which will close the include file and come into jakta command section with the command given after /. Usually ; return just goes to look after next command line.

Interactively testing some settings e.g. in regression splines can be done e.g. as follows.

put into iclude file e.g. commands:

```
group/spl etc...
;again:
;if(keystate.eq.1)incl io.in
;if(keystate.eq.1);goto again
```

If you see that group does not produce good looking spline, before clicking mouse open text file io.in with word pad, write as the last command new group command into it with prefix ;return/ (i.e ;return/group ...)and save the file but do not close it. Then go to figure window and press shift while clicking mouse. You will get the modified spline, and you can continue editing the definition by editing and saving io.in and pressing shift key when seeing the figure in the figure window.

corr/out=var !will save the covariance matrix for later use

- unsave commands for saving values of variables and objects (e.g. tables)

save/outfile=par.sav var_list ! save values of variables and objects into file par.sav

additional options:

/replace ! if outfile exists it is replaced, if the file exists and /replace is not given, error occurs

save command does not create versions for the outfile.

/close ! the outfile is closed, the default is that the file remains open, and one can save more objects into the file giving additional save commands without '/out=' option. save/close without var_list just closes the save file.

saved objects can be loaded with unsave command:

unsave/infile=par.sav var_list !load saved objects into variables given in var_list, if there is a file open for unsave, it is closed first

unsave var:list ! continue loading objects from the same file given in some previous unsave command in /infile= option.

unsave/close ! close file after unsave

Variable names are not saved, so the objects can be loaded under the same or different names. Objects are saved and unsaved one by one, so it is not necessary to load them in similar packages (var_-lists) as they are saved.

Both save and unsave have option /print which prints the type of object, and the number of parameters in the object.

Currently saving of lists does not work, because lists are objects consisting of variable indices and these are generally different if unsave is done after different session history.

May 2000

write-command has option /close

incl/wait/DELETEME described now here and works now also in Windows

If there is command

standv var_list

after jlpsim and before simu-command this means that the stand variables (c-variables in old JLP) are not taken from the observation matrix of Jaktá but are just read from the same file as tree data, i.e.. from the file determined by 'treed' command.

Plot level

If there is plot level in the data, then stand variables must contain variable 'nplots' which tells how many plots there are.

Plot variables read from data (the same file as tree data) are given by command

plotv var_list

First plot variable must be 'ntrees' which tells how many trees there are for each plot. It is assumed now that plot variables are on their own record (later e.g. MELA formats will be supported)

After each plot record there is the usual tree record corresponding to the plot in the same form as earlier when trees were just for the whole stand.

When there is plot variables there can be three different kinds of loops in the simulator:

for i=1,ntrees

...

end for

This is the same as earlier tree loop. It is it goes through all trees without considering plot level, and the transformations cannot utilize plot variables.

for i=1,nplot

...

end for

This loop goes just over plots, and can use and make plot variables.

for i=1,nplots

(plot transformations, initialization of e.g. tree level sums etc)

for j=1,ntrees

(tree transformations

end for

(plot transformations)

end for

Sorting can be done with respect to tree variables ignoring plot level, or plots can be sorted by doing sorting with respect to a plot variable, or trees can be sorted within each plot defining sorting with command:

treesortv=sort(treeavar)/plots

Thereafter the sorting can be used either in single treeloop over all trees (rarely useful) or tree loop within plot loop.

If plot and trees within plots are sorted, then nested loops can be done following both sorting orders, i.e.

```
for i=1,nplots/plotsort
```

```
for j=1,ntrees/treesort
```

If there is plot level, the new trees must be defined within plot loops.

sumfunctions cannot utilize plot loops yet.

```
*****end plot level
```

March 13, 2000:

sorting trees: search sort

March 1, 2000:

store and load in simulator: search store/load

Feb. 24..2000

general named transformations: search named transformation

matrix calculations: search: matrix calculations

generating matrix from data: search: data matrix

January 2000, recalculating cons-transformations: search cons/repeat

add to previous cons transformations: search cons/add

14.9.99. searchi: special write-function in the simulator

9.9.99 Search:keepperiodniin ko solmun alapuolelle ei generoida lisäsolmuja. Jos cons-käskyllä on annettu muuttujalle keepperiod pienempi arvo